Compiler Theory: Lexical Analysis

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2004

December 1, 2004

1 Finite Automata

1.1 Deterministic Finite Automata

ALPHABET, STRING, LANGUAGE. We call an *alphabet* any finite set of symbols. Let Σ be an alphabet. A *string* or *word* over Σ is any finite sequence of symbols from Σ. The empty string is denoted by ϵ. A *language* over Σ is any set of strings over Σ. Note that the above definitions do not ascribe any meaning to the strings of the language.

DETERMINISTIC FINITE AUTOMATA. A deterministic finite automaton (DFA) consists of five things:

- an input alphabet Σ,
- a finite set S whose elements are called *states*,
- a distinguished state $s_0 \in S$, called *starting state*,
- a set $F \subseteq S$ of distinguished states, called *accepting states* (or *final states*),
- a partial function δ from $S \times \Sigma$ to S called the *transition function* (hence for every state-symbol couple either δ maps it to a symbol or δ is not defined for this couple).

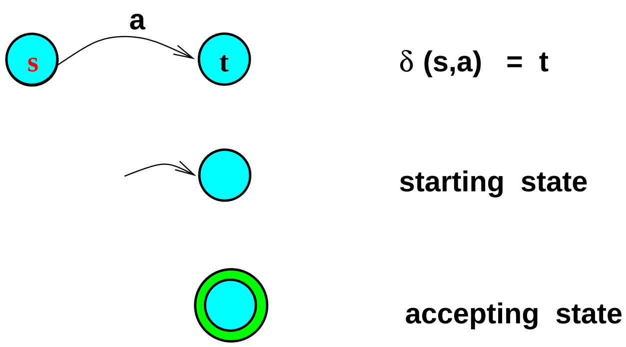


Figure 1: Pictorial notations for finite automata.

It is convenient

- to say "let $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$ be a DFA" for "let $\mathcal{A}$ be a DFA with alphabet Σ, set of states S, starting state s_0, set of accepting states F and transition function δ."
- to represent a DFA by a *transition diagram* using the rules shown on Figure 1.

LANGUAGE RECOGNIZED BY A DFA. A DFA accepts an input string if when beginning the computation in the starting state, after reading the entire string, the automaton is in an accepting state. The set of the strings recognized by a DFA $\mathcal{A}$ is called the *language recognized by the DFA* $\mathcal{A}$ and will be denoted by $\mathcal{L}(\mathcal{A})$.

TRANSITION TABLE. A straightforward way to implement a DFA is to represent the transition function δ as a *transition table.* This table has a row for each state s and a column for each input symbol a . The intersection of row s and column a contains $\delta(s, a)$. Figure 3 shows the transition diagram and the transition table of an automaton. A DFA can be easily converted to a program to look for tokens specified by it.

COMPLETE DFA. A DFA $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$ is said *complete* if for every $s \in S$ and for every $a \in \Sigma$ the transition $\delta(s, a)$ is defined.

The automaton of Figure 3 is complete whereas none of the automata of Figure 2 are.

As a model for a computer program, it is desirable for a DFA to be complete. To transform a non-complete DFA $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$ into a complete DFA one needs only

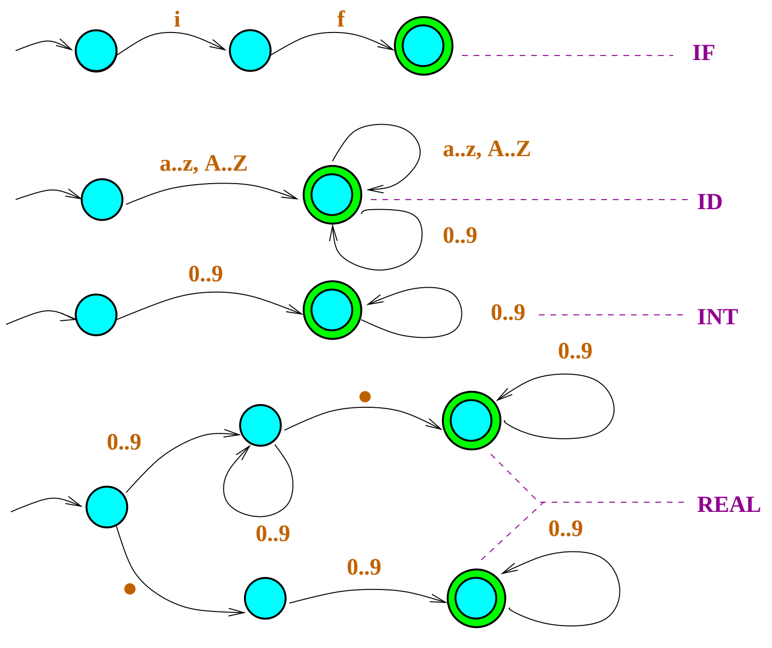
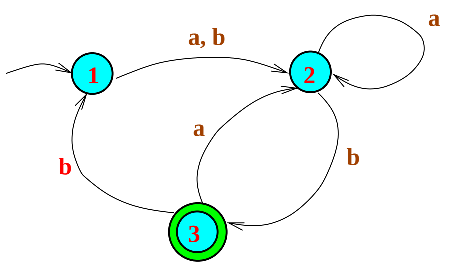


Figure 2: Some finite automata accepting some lexical tokens.



	a	b
1	2	2
2	2	3
3	2	1

Figure 3: Transition table for a DFA.

1. to add a new state to S, say E (for *error*) and then
2. for every $(s, a) \in S \times \Sigma$ such that δ was not defined at this couple, define $\delta(s, a) = E$.

<u>DFA recognizing the union of several languages.</u> Each of the four DFA in Figure 2 recognizes a simple language over the alphabet

$$\Sigma \;=\; [a \cdots z] \;\cup\; [A \cdots Z] \;\cup\; [0 \cdots 9] \;\cup\; \{.\} \tag{1}$$

Let us denote these languages by L_1, L_2, L_3, L_4.

- L_1 is the language reduced to the single world if,
- L_2 is the language of the identifiers made from letters and digits and starting with a letter,
- L_3 is the language of the integer numbers,
- L_4 is the language of the floating point numbers.

It is desirable to build a DFA that would recognize the language $L_1 \cup L_2 \cup L_3 \cup L_4$. In other words, one would like to combine the four transition diagrams in Figure 2 specifying the various patterns into a single transition diagram. This is a non-trivial task. Here are some difficulties.

- Consider a word w_i which belongs to some L_i and a word $w_j \neq \varepsilon$ such that $w_i w_j$ belongs to some L_j. For instance with $w_i = 123$ and $w_j = .456$ should the combined DFA accept w_i or $w_i w_j$? The rule is to choose the longest initial substring that can match a pattern (think of integers). Hence $w_i w_j$ is chosen.
- It may happen that $w_i w_j$ (the longest match) belongs to several L_j. Hence we need to define some priorities. For instance the word if belongs to L_1 and L_2. However we want the word if to be a reserved word, not an identifier. So we have to replace L_2 by $L_2 \setminus L_1$.

N.B. From now on and for simplicity, we do not consider capital letters any more in our example (Figure 2). In other words we reduce Σ to the union of $[a \cdots z]$, $[0 \cdots 9]$ and $\{.\}$.

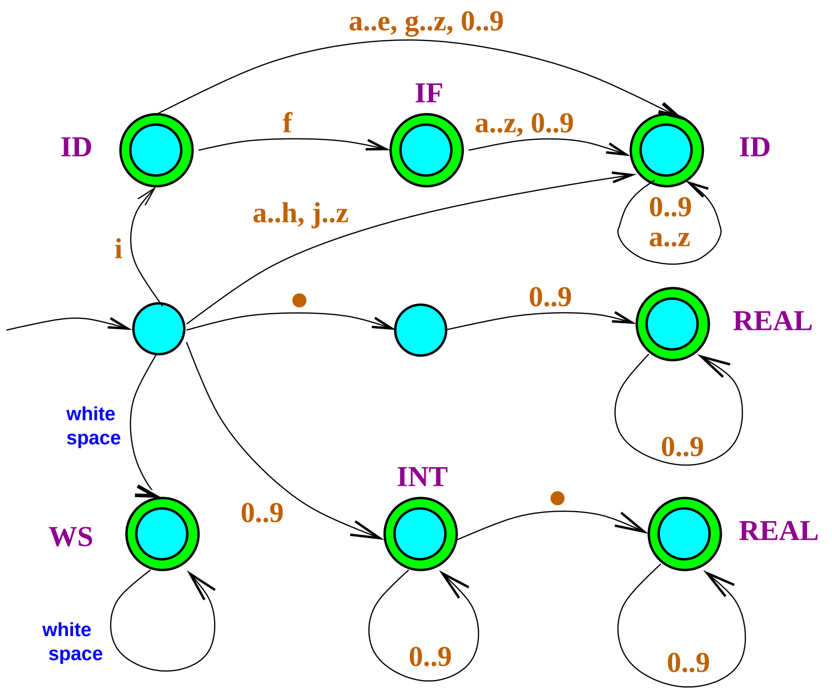


Figure 4: An automaton accepting $L_1 \cup L_2 \cup L_3 \cup L_4$ and solving the difficulties mentionned above.

1.2 Non-deterministic Finite Automata

<u>NON-DETERMINISTIC FINITE AUTOMATA.</u> A non-deterministic finite automaton (NFA) consists of five things:

- an input alphabet Σ,
- a finite set S whose elements are called *states*,
- a set $I \subseteq S$ of distinguished states, called *initial states*,
- a set $F \subseteq S$ of distinguished states, called *accepting states*,
- a function δ from $S \times \Sigma$ to 2^S, thus every state-symbol couple is mapped by δ to set of states (that is a subset of S).

Observe that

- In the transition graph of a NFA the same symbol $a \in \Sigma$ can label two or more transitions out of one state.
- Therefore after reading a given symbol, a NFA can have several states.

<u>LANGUAGE RECOGNIZED BY A NFA.</u> A word w is accepted by a NFA if after reading w the NFA is in at least one accepting state. Figure 5 shows a NFA and a DFA recognizing the same language: the language over $\Sigma = \{a, b\}$ consisting of the words which end with b.

<u>FROM A NFA TO A DFA ACCEPTING THE SAME LANGUAGE.</u>

Theorem 1 *Let $\mathcal{A} = (\Sigma, S, I, F, \delta)$ be a NFA accepting the language L. Then there exists a DFA $\overline{\mathcal{A}}$ accepting L too.*

To construct such a DFA $\overline{\mathcal{A}}$ from $\mathcal{A}$ one proceeds as follows.

1. The alphabet is unchanged.
2. The set $\overline{S}$ of states of $\overline{\mathcal{A}}$ is a subset of 2^S. Hence each state of $\overline{\mathcal{A}}$ is a subset of S. The set $\overline{S}$ is computed by Algorithm 1 below.
3. The starting state $\overline{I}$ of $\overline{\mathcal{A}}$ is the set I of the initial states of $\mathcal{A}$.
4. The set $\overline{F}$ of the accepting states of $\overline{\mathcal{A}}$ is the set of the states $\overline{s}$ of $\overline{S}$ such that $\overline{s} \cap F \neq \emptyset$.

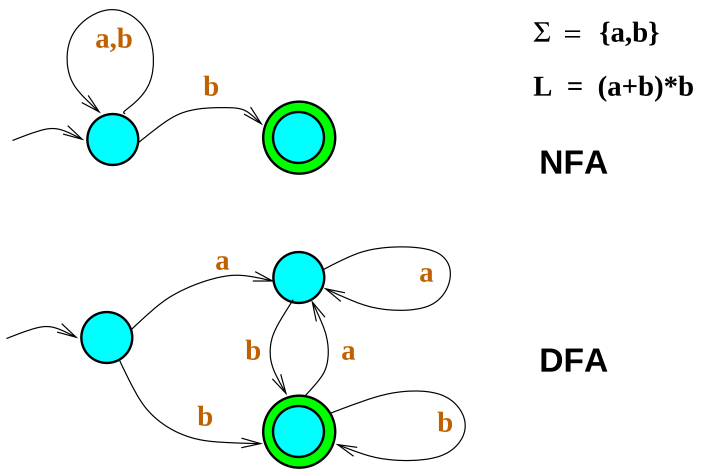


Figure 5: A NFA and a DFA accepting the same language.

5. The transition function Δ of $\overline{\mathcal{A}}$ together with $\overline{S}$ is determined by Algorithm 1, called the <u>SUBSET ALGORITHM</u>.

Algorithm 1

1 $toSee := [\overline{I}]$;

2 $\overline{S} := [\,]$;

3 **while** $toSee \neq [\,]$ **repeat**

  3.1 $A :=$ **first**$(toSee)$; $toSee :=$ **rest**$(toSee)$;

  3.2 **for** $x \in \Sigma$ **repeat**

    3.2.1 $\Delta(A, x) := \{s' \mid (\exists s \in A)\ (s \xrightarrow{x} s')\}$;

    3.2.2 **if** $(\Delta(A, x) \notin toSee)$ **and** $(\Delta(A, x) \notin \overline{S})$ **then** $toSee :=$ **cons**$(\Delta(A, x), toSee)$;

  3.3 $\overline{S} :=$ **cons**$(A, \overline{S})$;

4 **return**$(\overline{S}, \Delta)$

In this algorithm

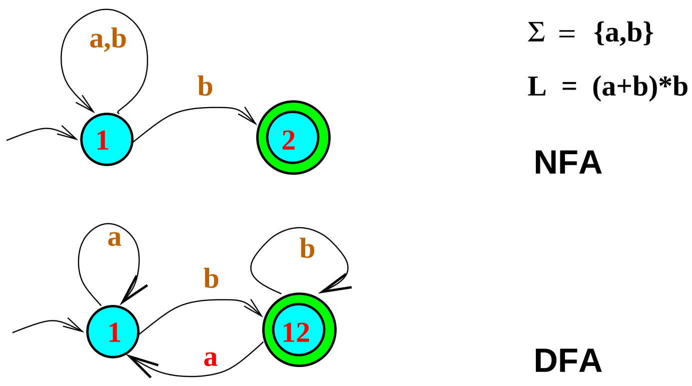


Figure 6: Another DFA accepting the same language as the previous NFA.

- *toSee* is the list of the states $\overline{s}$ of $\overline{\mathcal{A}}$ such that the $\Delta(a, \overline{s})$ for all $a \in \Sigma$ have not been computed yet,
- whereas $\overline{S}$ contains those states $\overline{s}$ of $\overline{\mathcal{A}}$ for which $\Delta(a, \overline{s})$ is known for every $a \in \Sigma$.
- the functions `first`, `rest` and `cons` operate on lists and are similar to the traditional stack routines `top`, `pop` and `push` respectively.

Figure 6 shows a DFA obtained from a NFA by applying Algorithm 1.

Remark 1 *States are generally denoted with numbers. So if the states of the input NFA are called* $1, 2, 3, \ldots$ *then the states of the output DFA could be* $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, $\{1, 2, 3\}$, $\ldots$. *To simplify notation we denote the subset* $\{i, j, \ldots\}$, *by* $ij\cdots$ *providing that there is no ambiguity.*

1.3 Non-deterministic Finite Automata with instantaneous transitions

NFA WITH INSTANTANEOUS TRANSITIONS. A non-deterministic finite automaton with instantaneous transitions (or ε-transitions) consists of six things:

- an input alphabet Σ,
- a finite set S whose elements are called *states*,
- a set $I \subseteq S$ of distinguished states, called *initial states*,

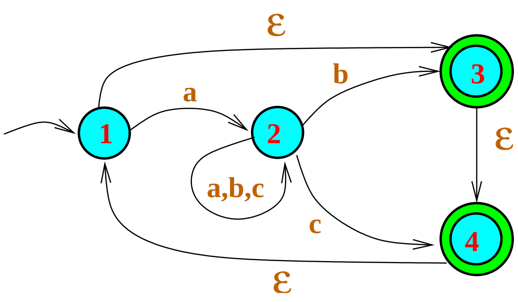


Figure 7: A non-deterministic finite automaton with instantaneous transitions.

- a set $F \subseteq S$ of distinguished states, called *accepting states*,
- a function δ from $S \times \Sigma$ to 2^S, thus every state-symbol couple is mapped by δ to a subset of S,
- a function τ mapping each state of S to a subset of S

such that $(\Sigma, S, I, T, \delta)$ is a NFA. The functions δ and τ are called the *non-instaneous transition function* and the *ε-transition function* (or the *instaneous transition function*). Such an automaton will be called a NFAIT or an ε-NFA, for short.

Instantaneous transitions.

Observe that

- The goal of the function τ is to describe transitions from one state to others without reading any letter. One can view also these ε-transitions from one state to others as transitions obtained after reading the empty word ε.
- In the transition graph of a NFAIT the ε-transitions are indicated by edges (from one state to another) labeled by ε.

The language accepted by a NFAIT. For $A \subseteq S$ we define the ε-closure of A as the set of the states that can be reached from a state of A by reading the empty word ε. The ε-closure of A is denoted by $\overline{\tau}(A)$ and is formally defined by Algorithm 2.

Algorithm 2

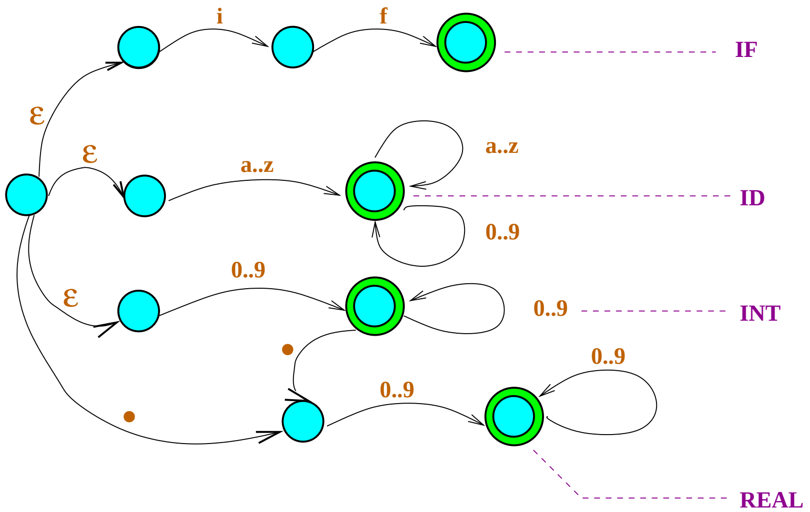


Figure 8: A NFAIT accepting $L_1 \cup L_2 \cup L_3 \cup L_4$.

1 $toSee := A$;

2 $\overline{\tau}(A) := A$;

3 **while** $toSee \neq \emptyset$ **repeat**

3.1 $s :=$ **first**$(toSee)$; $toSee :=$ **rest**$(toSee)$;

3.2 **for** $s' \in \tau(s)$ **repeat**

3.2.1 **if** $(s' \notin toSee)$ **and** $(s' \notin \overline{\tau}(A))$ **then** $toSee :=$ **cons**$(s', toSee)$;

3.3 $\overline{\tau}(A) :=$ **cons**$(s, \overline{\tau}(A))$;

4 **return**$(\overline{\tau}(A))$

For two states s and s', and for a word w we define the relation $s \xrightarrow{w} s'$ by induction on the length $|w|$ of w:

- if $w = \varepsilon$ we have $s \xrightarrow{w} s'$ for every $s' \in \overline{\tau}(\{s\})$,
- if $w = vx$ where v is a word and x is a letter, we have $s \xrightarrow{vx} s'''$ if there exist s', s'' such that $s \xrightarrow{v} s'$, $s'' \in \delta(s', x)$ and $s''' \in \overline{\tau}(\{s''\})$.

a word w over Σ is accepted (or recognized) by the NFAIT $\mathcal{A}$ if there exists an initial state i and a final state f such that $i \xrightarrow{w} f$.

FROM A NFAIT TO A DFA ACCEPTING THE SAME LANGUAGE.

Theorem 2 *Let $\mathcal{A} = (\Sigma, S, I, T, \delta, \tau)$ be a NFAIT and let L be the language accepted by $\mathcal{A}$. Then there exists a DFA $\overline{\mathcal{A}}$ which accepts L too.*

To construct such an $\overline{\mathcal{A}}$ one proceeds as in Algorithm 1 with the following two modifications.

- The initial state is $\overline{\tau}(I)$ instead of $\overline{I}$.
- The transition $\Delta(A, x)$ is defined by:

$$\Delta(A, x) \;=\; \overline{\tau}(\{s' \mid (\exists s \in A)\ (s' \in \delta(s, x))\}).$$

The resulting algorithm is called again the *subset algorithm.* Figure 9 shows a DFA accepting the same language as the NFAIT of Figure 7. This DFA is computed by means of the subset algorithm. Let us describe the construction of the transition table of this DFA.

1. The initial state is the $\overline{\tau}(I) = \overline{\tau}(1) = 134$. The transitions from it are $\Delta(134, a) = 2$, $\Delta(134, b) = \emptyset$ $\Delta(134, c) = \emptyset$. This shows that $\emptyset$ and 2 are states of the output DFA. The $\emptyset$ has to be understood as the *error-state*. If we do not need to produce a complete DFA then we can discard this *error-state*.

2. The transition from state 2 are $\Delta(2, a) = 2$, $\Delta(2, b) = 1234$ $\Delta(2, c) = 1234$. This shows that 1234 is a state of the output DFA.

3. The transition from state 1234 are $\Delta(1234, a) = 2$, $\Delta(1234, b) = 1234$ $\Delta(1234, c) = 1234$.

4. The final states of this DFA are its states which contain at least one of the final states of the input NFA. Hence the final states are 134 and 1234. Therefore we obtain the following transition table.

	a	b	c
134	2	$\emptyset$	$\emptyset$
2	2	1234	1234
1234	2	1234	1234

Remarks about the NFAIT-to-DFA construction.

- If the input NFAIT (of the subset algorithm) has q states then the output DFA has $\mathcal{O}(2^q)$ states.

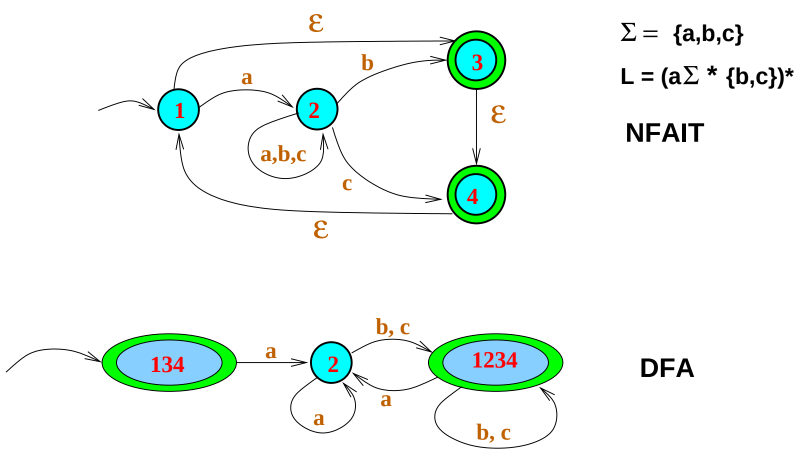


Figure 9: From a NFAIT to a DFA accepting the same language and produced by the subset algorithm.

- However the states of the DFA computed by this algorithm have the following property: they can all be reached on some input word. In other words, none of these computed states is useless.
- In practice, the number of reachable states (of the output DFA) is much smaller than the above exponential upper bound.
- However the DFA constructed by the above algorithm is not necessarily minimal: it is possible that the number of states can be further reduced. Figure 10 shows a DFA recognizing the same language as the DFA of Figure 4 but using 6 states instead of 9.

1.4 Finite Automata.

Because any non-deterministic finite automaton (with or without instantaneous transitions) can be replaced by a deterministic finite automaton recognizing the same language, from now on we will just say *finite automaton* (FA for short) instead of DFA, NFA or NFAIT. In practice, we will use

- NFAs or NFAITs for designing an automaton on the paper and then
- DFAs for implementing it on computer.

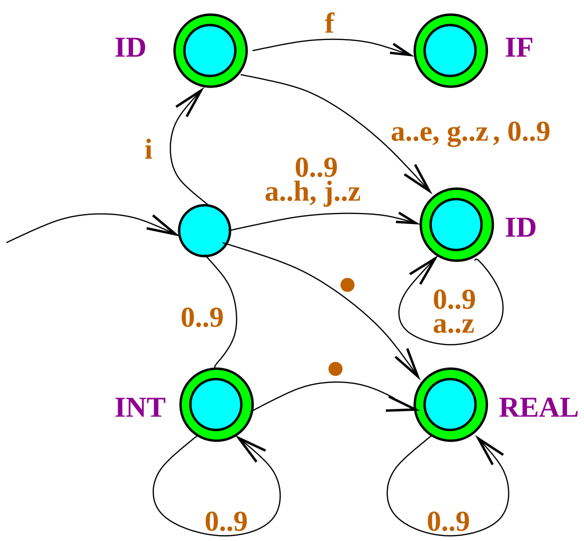


Figure 10: A minimal DFA accepting $L_1 \cup L_2 \cup L_3 \cup L_4$.

Indeed, NFAs and NFAITs are easier to manipulate but DFAs are closer to computer programs. Finally we will make the following simplification in the notations for NFAIT: we will denote the non-instaneous transition function and the ε-transition function with the same letter. More precisely, let $(\Sigma, S, I, T, \delta, \tau)$ be a NFAIT. We define for every $s \in S$

$$\delta(s, \varepsilon) \; = \; \tau(s) \tag{2}$$

With this convention we can give an unified definition for DFA, NFA or NFAIT.
<u>A finite automaton</u> over Σ is a 5-uple $(\Sigma, S, I, T, \delta)$ where

- S is a finite set, called the *set of states* of the FA,
- I anf T are non-empty subsets of S, called the sets of the *initial states* and *accepting states* of the FA, respectively,
- δ is a map from $S \times (\Sigma \cup \{\varepsilon\})$ to 2^S, called the transition function of the FA.

A non-empty word $w = x_1 x_2 \cdots x_n$ with length n is recognized by the FA $(\Sigma, S, I, T, \delta)$ if there exist states $s_1, s_2, \cdots, s_{n-1} \in S$ such that

1. the state s_1 is an initial state,

2. the state s_{i+1} belongs to $\delta(s_i, x_i)$ for $i = 1 \cdots n-1$,

3. the state s_{n-1} is an accepting state.

The empty word $w = \varepsilon$ is recognized by the above FA if

- either $I \cap F \neq \emptyset$
- or there exists an initial state s such that $\delta(s, \varepsilon)$ is an accepting state.

1.5 Languages recognized by finite automata

LANGUAGES RECOGNIZED BY FA. Let Σ be an alphabet. A language L over Σ is *recognized by FA* if there exists a finite automaton $\mathcal{A}$ such that L is the language recognized by $\mathcal{A}$.

THE PUNPING LEMMA. A natural question is

- are all languages over Σ recognized by FA?
- and if no can we characterize those languages which are recognized by FA?

To answer the first part of this question we start with the following remark.

Proposition 1 *Every language over Σ with a finite number of words is recognized by FA.*

Indeed let $L = \{w_1, w_2, \ldots, w_n\}$ be such a language. We can easily make FAs $\mathcal{A}_1$, $\mathcal{A}_1$, $\ldots \mathcal{A}_n$ recognizing the languages $L_1 = \{w_1\}$, $L_2 = \{w_1\}$, $\ldots$, $L_n = \{w_n\}$, respectively. Then by using ε-transitions we can construct a single FA $\mathcal{A}$ recognizing

$$L \quad = \quad L_1 \cup L_2 \cup \cdots \cup L_n. \tag{3}$$

Therefore, if a language L is not recognized by FA then it must contain an infinite number of words. This leads us to the following remark.

Proposition 2 *Let L be a language recognized by FA and with an infinite number of words. Let $\mathcal{A} = (\Sigma, S, I, T, \delta)$ be a DFA accepting L. Then $\mathcal{A}$ possesses a* circuit. *In other words, there exist a positive integer n together with n states $s_1, s_2, \ldots, s_n$ and $n-1$ letters $x_1, x_2, \ldots, x_{n-1}$ (not necessarily pairwise different) such that* (*i*) $s_{i+1} = \delta(s_i, x_i)$ *for every* $i = 1 \cdots n-1$,

(*ii*) $s_i \neq s_j$ *for* $i, j = 1 \cdots n-1$ *and* $i \neq j$,

(*iii*) $s_1 = s_n$.

Indeed, let s_1 be the initial state, q be the number of states of $\mathcal{A}$ and p the number of letters. Since L is infinite, there exists a word $w = x_1 \cdots x_\ell$ in L with lenght $\ell > q$. Recognizing w requires at least $q+1$ transitions:

$$s_2 = \delta(s_1, x_1), s_3 = \delta(s_3, x_3), \ldots s_\ell = \delta(s_{\ell-1}, x_{\ell-1}), s_{\ell+1} = \delta(s_\ell, x_\ell). \qquad (4)$$

At least two of the visited states s_i must be equal and the proposition is proved. Proposition 3 gives a more precise statement of the above statement and Figure 11 sketches its proof.

Proposition 3 (Pumping Lemma) *Let L be a language recognized by FA and with an infinite number of words. Then there exists a positive integer N such that for every word $m \in L$ with length $\ell > N$ there exist three words u, v, w such that*

(*i*) $m = u\,v\,w$,

(*ii*) $v \neq \varepsilon$,

(*iii*) *for every positive integer k we have $u\,v^k\,w \in L$.*

Example 1 *Let us apply Proposition 3 to the language $L = \{a^n b^n\}$ over $\Sigma = \{a, b\}$. Let us assume that L is recognized by FA. Let N, m, u, v, w be as in the proposition. If v counts more a than b then for k big enough $u\,v^k\,w$ will also have more a than b and thus cannot belong to L. Similarly if v counts more b than a then $u\,v^k\,w$ cannot belong to L for k big enough. So v must be of the form $a^n b^n$. But then for $k > 1$ the word $u\,v^k\,w$ does not belong to L. Finally we are led to a contradiction and the language L is recognized by FA.*

<u>Construction of languages recognized by FA</u> We would like now to characterize those languages which are recognized by FA. To address this question we give a series of four propositions and one theorem. Each of these four propositions provide a mechanism (or rule) to build languages recognized by FA. Theorem 3 states that these rules allow us to build all possible languages recognized by FAs.

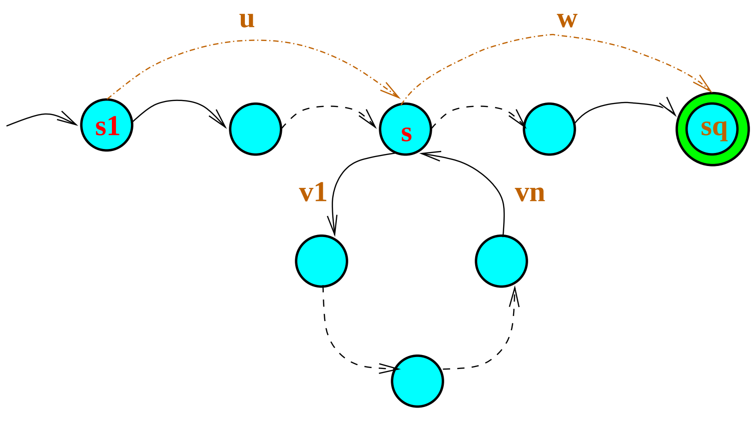


Figure 11: Sketch of the pumping Lemma.

- Proposition 4 expresses the fact that languages consisting of a single word consisting itself of a single letter are recognized by FA. This statement is an obvious consequence of the previous Proposition 1 and is illustrated by Figure 12.
- Proposition 5 expresses the fact that the union of two languages recognized by FA is itself recognized by FA. This statement is also trivial and was illustrated by Figure 4.
- Proposition 6 expresses the fact that the concatenation (or product) of two languages recognized by FA is also recognized by FA. This statement is also not difficult to prove and we have already used this result implicitely with most examples.
- Proposition 7 expresses the fact that the *star* of a language recognized by FA is also recognized by FA. The star (or *Kleene closure*) of a language is defined and illustrated below.

Proposition 4 *For every $x \in \Sigma \cup$ the language $L = \{x\}$ consisting of the single word $w = x$ is recognized by FA.*

To illustrate Proposition 5, 6 and 7 we will use the concept of a *normalized FA*.

<u>Normalized Finite Automata</u> A finite automaton $\mathcal{A} = (\Sigma, S, I, F, \delta)$ is *normalized* if it satisfies to the following properties

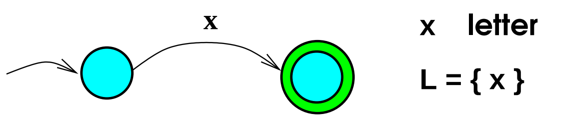


Figure 12: A DFA accepting a language consisting of a single letter.

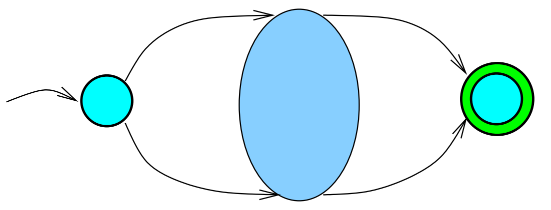


Figure 13: A normalized FA.

- $\mathcal{A}$ has only one initial state, say s,
- $\mathcal{A}$ has only one final (= accepting) state, say f,
- no transition leads to the initial state s, or more formally, for every $t \in S$ and every $x \in \Sigma \cup \{\varepsilon\}$ we have $s \notin \delta(t, x)$,
- no transition leaves from the final state f, or more formally, for every $x \in \Sigma \cup \{\varepsilon\}$ we have $\delta(f, x) = \emptyset$.

Normalized FA are generally depicted as shown on Figure 13.

<u>The union of two languages recognized by FA</u> is a language recognized by FA. Proposition 5 formulates this statement with DFAs and Figure 15 illustrates it with normalized FAs.

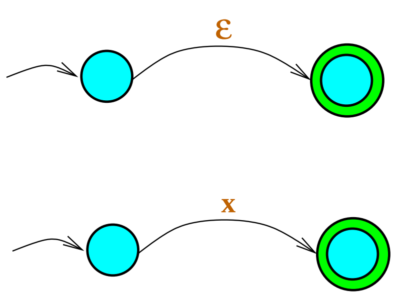


Figure 14: Normalized FAs for R0 languages.

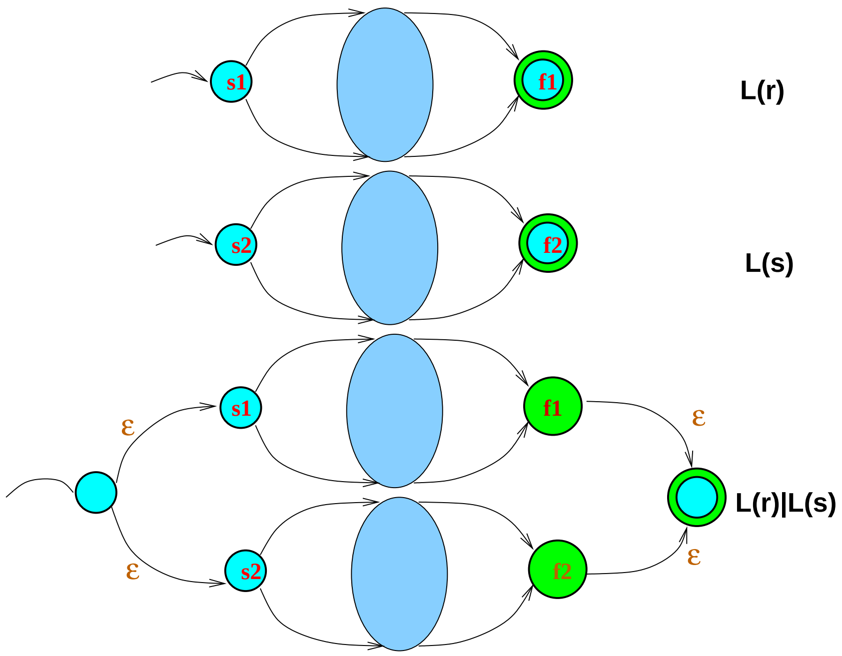


Figure 15: A normalized FA accepting the sum of two other normalized FAs.

Proposition 5 *Let L_1 and L_2 be two languages over Σ recognized by the DFAs $\mathcal{A}_1 = (\Sigma, S_1, s_1, F_1, \delta_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, s_2, F_2, \delta_2)$ respectively. Let us assume that $S_1 \cap S_2 = \emptyset$. (If this is not the case then we can* rename *the states of $\mathcal{A}_2$.)*

Then, the language $L_1 \cup L_2$ is recognized by the NFA

$$\mathcal{A}_{1+2} = (\Sigma, S_1 \cup S_2, \{s_1, s_2\}, F_1 \cup F_2, \delta_{1+2}) \tag{5}$$

where the transition function δ_{1+2} is defined as follows for every $x \in \Sigma$ and for every $s \in S_1 \cup S_2$

$$\delta_{1+2}(s, x) \;=\; \begin{cases} \delta_1(s, x) & \text{if} \quad s \in S_1 \\ \delta_1(s, x) & \text{if} \quad s \in S_2. \end{cases} \tag{6}$$

<u>THE PRODUCT OF TWO LANGUAGES RECOGNIZED BY FA</u> is a language recognized by FA. Proposition 6 formulates this statement with DFAs and Figure 16 illustrates it with normalized FAs.

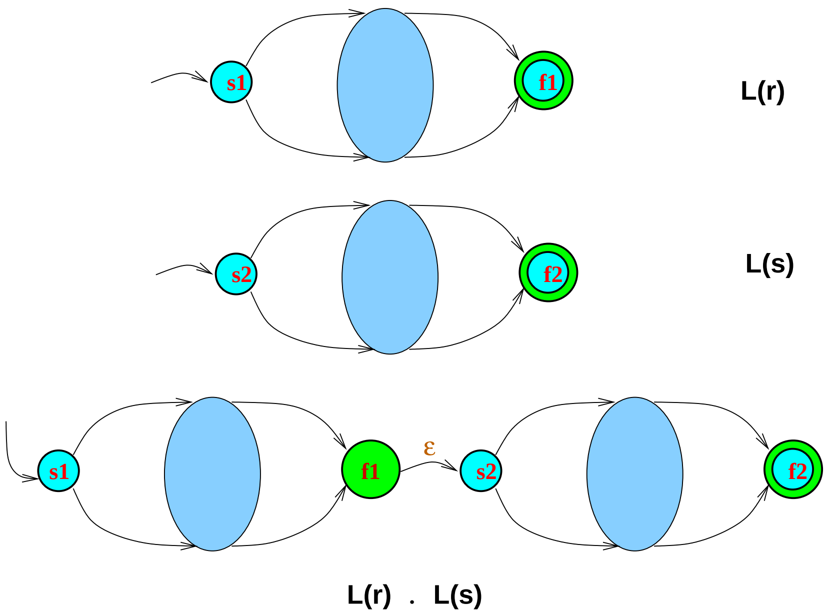


Figure 16: A normalized FA accepting the sum of two other normalized FAs.

Proposition 6 *Let L_1 and L_2 be two languages over Σ recognized by the DFAs $\mathcal{A}_1 = (\Sigma, S_1, s_1, F_1, \delta_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, s_2, F_2, \delta_2)$ respectively. Again, let us assume that $S_1 \cap S_2 = \emptyset$.*

We denote by L_1L_2 (or $L_1.L_2$) the language consisting of all words of the form w_1w_2 (concatenation of w_1 and w_2) where w_1 and w_2 belong to L_1 and L_2 respectively. This language is called the PRODUCT LANGUAGE *of L_1 by L_2*

Then, the language L_1L_2 is recognized by the NFA

$$\mathcal{A}_{1.2} = (\Sigma, S_1 \cup S_2, \{s_1\}, F_2, \delta_{1.2}) \tag{7}$$

where the transition function $\delta_{1.2}$ is defined as follows for every $x \in \Sigma \cup \{\varepsilon\}$ and for every $s \in S_1 \cup S_2$

$$\delta_{1.2}(s, x) = \left\{ \begin{array}{rl} \delta_1(s, x) & \text{if} \quad s \in (S_1 \setminus F_1) \\ \delta_1(s, x) & \text{if} \quad s \in F_1 \quad \text{and} \quad x \neq \varepsilon \\ \{s_2\} \cup \delta_1(s, x) & \text{if} \quad s \in F_1 \quad \text{and} \quad x = \varepsilon \\ \delta_2(s, x) & \text{if} \quad s \in S_2. \end{array} \right. \tag{8}$$

KLEENE CLOSURE OF A LANGUAGE. Let Σ be an alphabet and let w be a word over Σ. Let n be a non-negative integer. First we define the n-th power

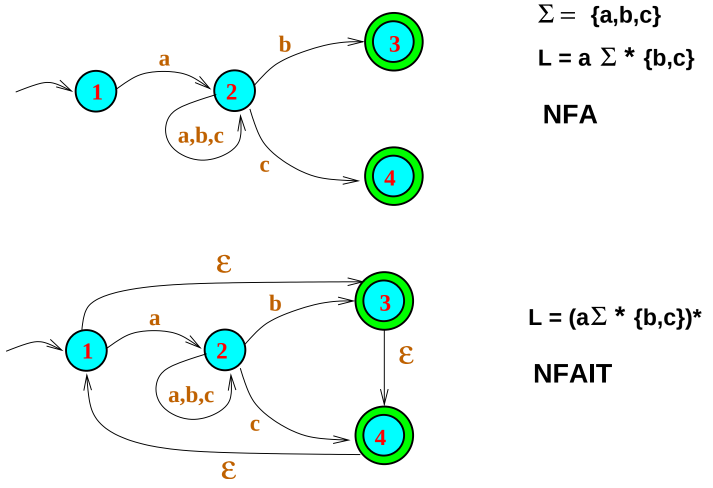


Figure 17: A DFA accepting a language L and a NFAIT accepting the star of L.

of w, denoted by w^n, as follows:

$$w^n \;=\; \left\{ \begin{array}{rll} \varepsilon & \text{if} & n = 0 \\ w.w^{n-1} & \text{if} & n > 0. \end{array} \right. \tag{9}$$

Let L be a language over Σ. Now we define the n-th power of L, denoted by L^n, as follows:

$$L^n \;=\; \left\{ \begin{array}{rll} \{\varepsilon\} & \text{if} & n = 0 \\ L.L^{n-1} & \text{if} & n > 0. \end{array} \right. \tag{10}$$

Then we define the *star* (or *Kleene closure*) of L, denoted by L^*, as the union of all powers of L, that is

$$L^* \;=\; \bigcup\nolimits_{n \geq 0} L^n. \tag{11}$$

Therefore, a word w over Σ belongs to L^* if

- either w is the empty word ε
- or there exists a positive integer n and a word u such that $w = u^n$.

We give now some useful formulas. Let L and M be two languages over Σ. Observe that

- Σ^* is the set of all words over Σ.

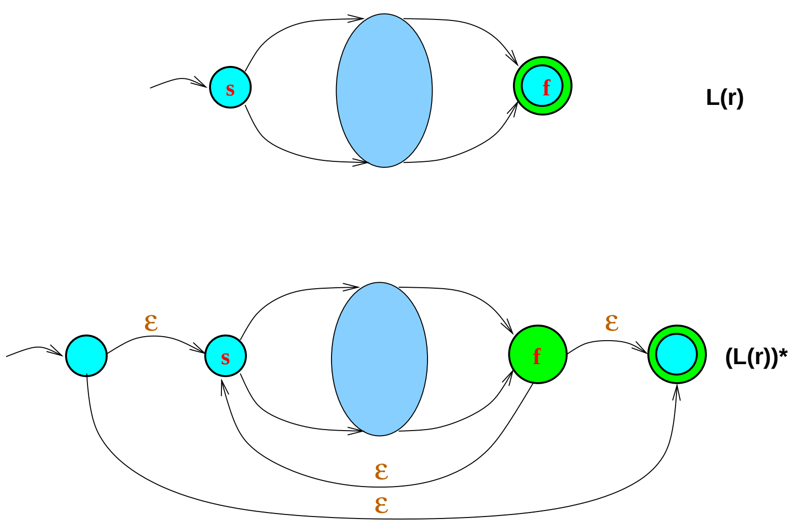


Figure 18: A normalized FA accepting the star of another normalized FA.

- Let L^+ be the union of all L^n with $n > 0$. Then $L^+ = L.L^*$.
- If $L \subseteq M$ then $L^* \subseteq M^*$.
- For every non-negative integer n we ahve $L^{*n} = L^*$.
- $L^{**} = L^*$.
- $(L^* + M^*)^* = (L^*.M^*)^*$.
- $(L^* + M^*)^* = (L + M)^*$.

The Kleene closure of a language recognized by FA is a language recognized by FA. Proposition 7 formulates this statement with DFAs and Figure 18 illustrates it with normalized FAs.

Proposition 7 *Let L be a language over Σ recognized by the DFA $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$.*

Then L^* is recognized by the NFA

$$\mathcal{A}^* = (\Sigma, S, s, F, \delta^*) \tag{12}$$

where the transition function δ^* is defined as follows for every $x \in \Sigma \cup \{\varepsilon\}$ and for every $s \in S$

$$\delta^*(s,x) \;=\; \begin{cases} \delta(s,x) & \text{if} \quad x \neq \varepsilon \\ F & \text{if} \quad s = s_0 \quad \text{and} \quad x = \varepsilon \\ \{s_0\} & \text{if} \quad s \in F \quad \text{and} \quad x = \varepsilon \end{cases} \tag{13}$$

<u>KLEENE THEOREM.</u> We denote by $R_0(\Sigma)$ the set of all languages of the form $\{x\}$ for $x \in \Sigma \cup \{\varepsilon\}$. Then for a positive integer n we define $R_n(\Sigma)$ as the set of languages over Σ consisting of

1. the languages of $R_{n-1}(\Sigma)$,
2. the languages of the form $L + M$ for $L, M \in R_{n-1}(\Sigma)$,
3. the languages of the form $L.M$ for $L, M \in R_{n-1}(\Sigma)$,
4. the languages of the form L^* for $L \in R_{n-1}(\Sigma)$.

Obviously, for every non-negative integer n, every language member of $R_n(\Sigma)$ is recognized by FA. The theorem of Kleene studies the *opposite direction.*

Theorem 3 (Kleene) *Let L be a language recognized by FA. Then there exists a non-negative integer n such that L belongs to $R_n(\Sigma)$.*

This suggests the introduction of the notion of a *regular expression.*

1.6 Regular expressions

<u>REGULAR EXPRESSION.</u> Let Σ be an alphabet such that none of the symbols $($, $)$, $|$, * belongs to Σ. A *regular expression* over Σ

- is a word r over the alphabet $\Sigma \cup \{(,), |, ^*\}$,
- denoting (= associated with) a language $L(r)$ over Σ and
- defined by the following inductive rules.

(R1) The empty word ε is a regular expression over Σ denoting the language $\{\varepsilon\}$.

(R2) For every $x \in \Sigma$ the letter x is a regular expression over Σ denoting the language $\{x\}$.

(R3) Let r and s be regular expressions over Σ denoting languages $L(r)$ and $L(s)$ respectively. Then,

- $*$ $(r)|(s)$ is a regular expression over Σ denoting the language $L(r) \cup L(s)$,
- $*$ $(r)(s)$ is a regular expression over Σ denoting the language $L(r).L(s)$,
- $*$ $(r)^*$ is a regular expression over Σ denoting the language $(L(r))^*$,
- $*$ (r) is a regular expression over Σ denoting the language $L(r)$ (i.e., one can add parentheses).

Remark 2 *Unnecessary parentheses can be avoided in regular expressions if we adopt the following conventions:*

1. *the unary operator * has the highest precedence and is left associative,*
2. *concatenation has the second the highest precedence and is left associative,*
3. *the operator $|$ has the lowest precedence and is left associative.*

With these conventions the regular expression $(a)|((b) * (c))$ is $a|b * c$.

<u>REGULAR LANGUAGES.</u> Let Σ be an alphabet (such that none of the symbols (,), |, * belongs to it). A language $\mathcal{L}$ over Σ is *regular* if there exists a regular expression r over Σ denoting $\mathcal{L}$, that is if $\mathcal{L} = L(r)$.

It follows from Kleene's theorem that a language is recognized by FA iff it is regular.

<u>FROM A REGULAR EXPRESSION TO A FINITE AUTOMATON.</u> We present now a method that for any regular expression r constructs a FA $\mathcal{A}(r)$ recognizing $L(r)$.

- The FA $\mathcal{A}(r)$ is built inductively by following the structure of the regular expression r.
- At each step of the algorithm a FA is produced from previous ones starting with FA of $R_0(\Sigma)$ and ending with $\mathcal{A}(r)$.

- Every intermediate FA $\mathcal{A} = (\Sigma, S, I, F, \delta)$ produced by the algorithm is NORMALIZED.

The method is quite natural.

1. We first parse r into its constituent subexpressions.
2. We construct NFAs for each letter or ε in r by applying the above rules (R1) and (R2).
3. Then guided by the syntaxic structure of the regular expression r, we combine the NFAs using rules (R3) and Figures 15, 16, 18.

This method is called *Thompson's algorithm*.

Remark 3 *The number of states of the FA produced by the Thompson's algorithm is upper bounded by* $2|r|$ *where* $|r|$ *is the number of symbols and operators in the regular expression* r*.*

2 Lexical Analysis

2.1 LEX

2.1.1 Introduction

DESCRIPTION. LEX is a tool for generating programs that perform pattern-matching on text. More precisely LEX reads

- the given input files
- or its standard input if no file names are egiven,

for a description of a scanner to generate in form of a C source file, `lex.yy.c`, which defines a routine `yylex()`.

FORMAT OF THE INPUT FILE The flex input file consists of three sections, separated by a line with just `%%` in it:

```
definitions
%%
rules
%%
user code
```

- Any section may be empty.
- Nonempty sections must be separated from each other by %%.
- The separator between the first and the second sections is necessary.

Example 2 *The following flex input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name.*

```
[moreno@iguanodon lex]$ more username.l
%%
username    printf( "%s", getlogin() );
[moreno@iguanodon lex]$ lex username.l
[moreno@iguanodon lex]$ gcc lex.yy.c  -ll -o username.exe
[moreno@iguanodon lex]$  ./username.exe
username
moreno
user name
user name
```

By default, any text not matched by a flex scanner is copied to the output.

Example 3 *This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals,* `num_lines` *and* `num_chars`*, which are accessible both inside* `yylex()` *and in the* `main()` *routine declared after the second %%. There are two rules. One which matches a newline and increments both the line count and the character count. The other one which matches any character other than a newline (indicated by the* `.` *regular expression).*

```
[moreno@iguanodon lex]$ more count.l
        int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
```

```
     {
     yylex();
     printf( "# of lines = %d, # of chars = %d\n",
            num_lines, num_chars );
}
[moreno@iguanodon lex]$ flex count.l
[moreno@iguanodon lex]$ gcc lex.yy.c -ll -o count.out
[moreno@iguanodon lex]$ ./count.out < count.l
# of lines = 13, # of chars = 222
```

Example 4 `[moreno@iguanodon lex]$ more trivialLanguage.l`

```
%{
/* need this for the call to atof() below */
#include <stdlib.h>
%}

DIGIT    [0-9]
ID       [a-zA-z][a-zA-z0-9]*
INT      {DIGIT}+
REAL     {DIGIT}*"."{DIGIT}+


%%

{INT}       {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{REAL}      {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if          {printf( "The keyword: %s\n", yytext );
            }

{ID}        printf( "An identifier: %s\n", yytext );
```

```
[ \t\n]+          /* eat up whitespace */

.           printf( "Unrecognized character: %s\n", yytext );
```

```
[moreno@iguanodon lex]$ flex trivialLanguage.l
[moreno@iguanodon lex]$ gcc lex.yy.c -ll -o trivialLanguage.out
[moreno@iguanodon lex]$ ./trivialLanguage.out
123.456 if compiler ok then x := -780000000000000000000000000000000000
A float: 123.456 (123.456)
The keyword: if
An identifier: compiler
An identifier: ok
An identifier: then
An identifier: x
Unrecognized character: :
Unrecognized character: =
Unrecognized character: -
An integer: 780000000000000000000000000000000000 (2147483647)
```

<u>The definition section.</u> Each line in this section has the general form

`name expression`

where

- `name` shoud begin in the first column
- there is at least one blank or tabulation between `name` and `expression`.

Moreover anything global to `lex.yy.c`

- is usually declared in this section
- between lines containing `%{` and `%}`
- and is copied literally to `lex.yy.c`.

<u>Translation rules section.</u> Each line in this section has the general form

```
expression action
```

where

- `action` is a C statement or whatever you want echoed. Unmatched input is echoed directly to output.
- `expression` is a LEX regular expression.

<u>User code.</u> In this section the user

- can redefine the `main()` function whose default is

```
int main()
        {
        yylex();
        return 0;
        }
```

- can redefine the `yywrap()` function which is called when the LEX generated program reads the end-of-file,
- may define other routines which are called in the translation rules section.

2.1.2 LEX regular expressions

<u>LEX regular expressions.</u> A LEX regular expression is a word made of

- text characters (letters of the alphabet, digits, ...)
- operators : `" \ { } [ ] ^ $ < > ? . * + | () /`

Moreover

- An operator can be used as a text character if it preceeded with the escape operator (backslash).
- The quotation marks indicate that whatever is contained between a pair of quotes is to be taken as text characters. For instance

```
xyz"++"
```

matches the string `xyz++`.

<u>A CHARACTER CLASS</u> is a class of characters specified using the operator pair `[ ]`. The expression

```
[ab]
```

matches the string `a` or `b`.

Within square brackets most operators are ignored except the three special characters `\ - ^` are which used as follows

(a) the escape character `\` as above,

(b) the minus character `-` which is used for ranges like in

```
digit        [0-9]
```

(c) the *hat* character `^` as first character after the opening square bracket, it is used for complemented matches like in

```
NOTabc       [^abc]
```

<u>OPTIONAL EXPRESSIONS.</u> The `?` operator indicates an optional element of an expression. For instance

```
ab?c
```

matches either `ac` or `abc`.

<u>REPEATED EXPRESSIONS.</u> Repetitions of patterns are indicated by the operators `*` and `+`.

- The pattern `a*` matches any number of consecutive `a` characters (including zero).
- The pattern `[a-z]+` is any positive number of consecutive lower-case alphabetic characters.

Hence we can recognize identifiers in a typical computer language with

```
[A-Za-z][A-Za-z0-9]*
```

Repetitions can also be obtained with the pair operator {}.

- If {} encloses numbers, it specifies repetitions. For instance `a{1,5}` matches 1 to 5 repetitions of `a`.
- **Note** that if {} encloses a name, this name should be defined in the definition section. Then LEX substitutes the definition for the name.

ALTERNATING. The operator `|` indicates alternation. For instance

```
(ab|cd)
```

matches the language consisting of both words `ab` and `cd`.

GROUPING. Parentheses are used for grouping (when not clear). For instance

```
(ab|cd+)?(ef)*
```

denotes the language of the words that are either empty or

- optionally starts with
  - `ab` or
  - `c` followed by any positive number of `d`
- and continues with any number of repetition of `ef`

Another example: an expression specifying a real number

```
-?(([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
```

where `\.` denotes a literal period.

CONTEXT SENSITIVITY. LEX provides some support for contextual grammatical rules.

- If `^` is the first character in an expression, then this expression will only be matched at the beginning of a line.
- If `$` is the last character in an expression, then this expression will only be matched at the end of a line.
- If r and s are two LEX regular expressions then r`/`s is another LEX regular expression.

- It matches r if and only if it is followed by an s.
- It is called a *trailing context*.
- After use in this context, s is then returned to the input before the action is executed. So the action only sees the text matched by r
- *Left context* is handled by means of *start conditions* which we will talk about later.

2.1.3 How the input is matched

Basic principles.

(1) When the generated scanner is run, it analyses its input string looking for strings which match any of its patterns.

(2) If the current input can be matched by several expressions, then the ambiguity is resolved by the following rules.

  (2.1) The longest match is preferred.

  (2.2) The rule given first is preferred.

(3) Once the match is determined,

  (3.1) the text corresponding to it is available in the global character pointer `yytext` its length is `yyleng` and the current line number is `yylineno`,

  (3.2) and the action corresponding to the matched pattern is then executed,

  (3.3) and then the remaining input is scanned for another match.

LEX routines and macros. There are a number of special directives which can be included within an action.

`ECHO` copies `yytext` to the scanner's output.

`REJECT` directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input). For example, the following will both count the words in the input and call the routine special() whenever "foo" is seen:

```
int word_count = 0;
%%
foo     special(); REJECT;
[^ \t\n]+   ++word_count;
```

Multiple `REJECT`'s are allowed, each one finding the next best choice to the currently active rule.

`yymore()` tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of yytext rather than replacing it. The following

```
%%
very[ ] {ECHO; yymore();}
good ECHO;
```

will write "very very good" to the output.

See also `yyless()`, `yyterminate()`, `input()`, `output()`, `unput()` in the documentation.

<u>Start conditions</u> (or start conditions) are used to limit the scope of certain rules or to change the way the lexer treats part of the file.

- Start conditions are declared in the definition section. For instance

  ```
  %s SC
  ```

  creates a start condition called `SC`.

- In the transition rules section, a rule of the form

  ```
  <SC>[aeiouy]* ECHO;
  ```

  will only apply in the state `SC`.

- A state is changed with a statement

  ```
  BEGIN SC;
  ```

  in an action.

- LEX starts in state zero (0), also called initial state.

  ```
  BEGIN 0;
  ```

  will change back to this initial state.

- Rules that do not have start states can be applied in any state.

2.1.4 More examples

<u>A french-to-javanese translater.</u>

```
[moreno@iguanodon lex]$ more src/french2javanese.l
%s CR
VOWEL       [Aa]|[Ee]|[Ii]|[Oo]|[Uu]|[Yy]
CONSONANT   [B-Db-dF-Hf-hJ-Nj-nP-Tp-tV-Xv-xZz]

%%

<CR>{VOWEL}        {BEGIN 0; printf("%s","av"); ECHO;}
{CONSONANT}        {ECHO; BEGIN CR;}
%%
[moreno@iguanodon lex]$ make french2javanese.out
[moreno@iguanodon lex]$ ./bin/french2javanese
Bonjour chers etudiants
Bavonjavour chavers avetavudaviants
Est-ce que le cours est interessant aujourd'hui ?
avEst-cave qavue lave cavours avest avintaveravessavant avaujavourd'havui ?
```

Observe that the last line above starts with `avEst`. Can you explain why ?

<u>An english-to-french translater for the Pascal language.</u> Our aim is also to

- remove extra white space but maintain identation,
- capitalize keywords.

```
[moreno@iguanodon lex]$ more test/pascal2pascal.test
program p;

var i,j : integer;

begin
   i := 23;
   j := i + i;
end.

[moreno@iguanodon lex]$ more src/pascal2pascal.l
%{
/* need this for the call to atof() below */
#include <math.h>
#include <stdio.h>

FILE *fic;


%}

PLUS        [\+]
MINUS       [\-]
TIMES       [\*]
BY          [/]
ASSIGN      :=
BINARYOP    {PLUS}|{MINUS}|{TIMES}|{BY}|{ASSIGN}
SEPARATOR   [,:;\.]
PROGRAM     [Pp][Rr][Oo][Gg][Rr][Aa][Mm]
VAR         [Vv][Aa][Rr]
IF          [Ii][Ff]
THEN        [Tt][Hh][Ee][Nn]
ELSE        [Ee][Ll][Ss][Ee]
BEGIN       [Bb][Ee][Gg][Ii][Nn]
```

```
END         [Ee][Nn][Dd]
WHILE       [Ww][Hh][Ii][Ll][Ee]
FOR         [Ff][Oo][Rr]
DO          [Dd][Oo]
TO          [Tt][Oo]
FUNCTION    [Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn]
PROCEDURE   [Pp][Rr][Oo][Cc][Ee][Dd][Uu][Rr][Ee]
LETTER      [A-Za-z]
DIGIT       [0-9]
ID          {LETTER}({LETTER}|{DIGIT})*
INTEGER     ({PLUS}|{MINUS})?{DIGIT}+
REAL        {INTEGER}(\.({DIGIT})*)?([eE]{INTEGER})?
COMMENT     "{"[^}\n]*"}"
%%

{INTEGER}   {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            fprintf(fic, "%d", atoi(yytext));
            }

{REAL}      {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            fprintf(fic, "%g", atof(yytext));
            }

{IF}        convert_IF();

{THEN}      convert_THEN();

{ELSE}      convert_ELSE();

{BEGIN}     convert_BEGIN();

{END}       convert_END();

{VAR}       convert_VAR();
```

```
{FOR}       convert_FOR();

{DO}        convert_DO();

{TO}        convert_TO();

{WHILE}     convert_WHILE();

{PROCEDURE} convert_PROCEDURE();

{FUNCTION}  convert_FUNCTION();

{PROGRAM}   convert_PROGRAM();

{ID}                        {
            printf( "An identifier: %s\n", yytext );
            fprintf(fic, "%s", yytext );
            }

{SEPARATOR} {
           printf( "A separator: %s\n", yytext );
           fprintf(fic, "%s", yytext );
           }

{BINARYOP}   {
           printf( "A binary operator: %s\n", yytext );
           fprintf(fic, "%s", yytext );
           }

{COMMENT}     {
                  printf( "A comment : %s\n", yytext );
                  fprintf(fic, "%s", yytext );
           }

^[ \t]+          fprintf(fic, "%s", yytext );
[ \t]+           fprintf(fic, "%s", " ");
[\n]+            fprintf(fic, "%s", "\n");
```

```
.           printf( "Unrecognized character: %s\n", yytext );

%%

keyword_message() { printf( "A keyword: %s\n", yytext );}

convert_IF() { keyword_message(); fprintf(fic, "%s", "SI"); }
convert_THEN() { keyword_message(); fprintf(fic, "%s", "ALORS"); }
convert_ELSE() { keyword_message(); fprintf(fic, "%s", "SINON"); }
convert_BEGIN() { keyword_message(); fprintf(fic, "%s", "DEBUT") ; }
convert_END() { keyword_message(); fprintf(fic, "%s", "FIN") ; }
convert_VAR() { keyword_message(); fprintf(fic, "%s", "VAR") ; }
convert_FOR() { keyword_message(); fprintf(fic, "%s", "POUR") ; }
convert_DO() { keyword_message(); fprintf(fic, "%s", "FAIRE") ; }
convert_TO() { keyword_message(); fprintf(fic, "%s", "A") ; }
convert_WHILE() { keyword_message(); fprintf(fic, "%s", "TANT QUE") ; }
convert_PROCEDURE() { keyword_message(); fprintf(fic, "%s", "PROCEDURE") ; }
convert_FUNCTION() { keyword_message(); fprintf(fic, "%s", "FONCTION") ; }
convert_PROGRAM() { keyword_message(); fprintf(fic, "%s", "PROGRAMME") ; }




int main() {

     fic = fopen("/tmp/pascal","w");

     yylex();

     fclose(fic);
     return 0;
}


[moreno@iguanodon lex]$ more test/pascal2pascal.test
```

```
program p;

var i,j : integer;

begin
   i := 23;
   j := i + i;
end.

moreno@iguanodon lex]$ ./bin/pascal2pascal < test/pascal2pascal.test
A keyword: program
An identifier: p
A separator: ;
A keyword: var
An identifier: i
A separator: ,
An identifier: j
A separator: :
An identifier: integer
A separator: ;
A keyword: begin
n identifier: i
A binary operator: :=
An integer: 23 (23)
A separator: ;
An identifier: j
A binary operator: :=
An identifier: i
A binary operator: +
An identifier: i
A separator: ;
A keyword: end
A separator: .
[moreno@iguanodon lex]$ more /tmp/pascal
PROGRAMME p;
VAR i,j : integer;
DEBUT
   i := 23;
```

```
   j := i + i;
FIN.
```

A SCANNER FOR C/JAVA SOURCE FILES We aim to count the number of

- comment lines without source code
- white lines
- lines containing source code.

```
[moreno@iguanodon lex]$ more src/C_source_scanner.l
%{
int comments = 0, code = 0, whiteSpace = 0;
%}

%s CS

COMMENT             "/*".*"*/"
COMMENTLINE         ^[ \t]*{COMMENT}[ \t]*\n
WHITELINE           ^[ \t]*\n
CODESTARTINGLINE    .+({COMMENT}.+)*.*\n
CODEENDINGLINE      .*({COMMENT}.+)*.+\n
CODELINE            {CODESTARTINGLINE}|{CODEENDINGLINE}

STARTCOMMENTLINE    ^[ \t]*"/*"

%%

{COMMENTLINE}       {comments++;
                    /* comment on a single line */
                    }
{WHITELINE}         {whiteSpace++;}
{CODELINE}          {code++;}

{STARTCOMMENTLINE} {BEGIN CS;
                    /* enter comment eating state */
                    }
<CS>"*/"        {BEGIN 0;}
```

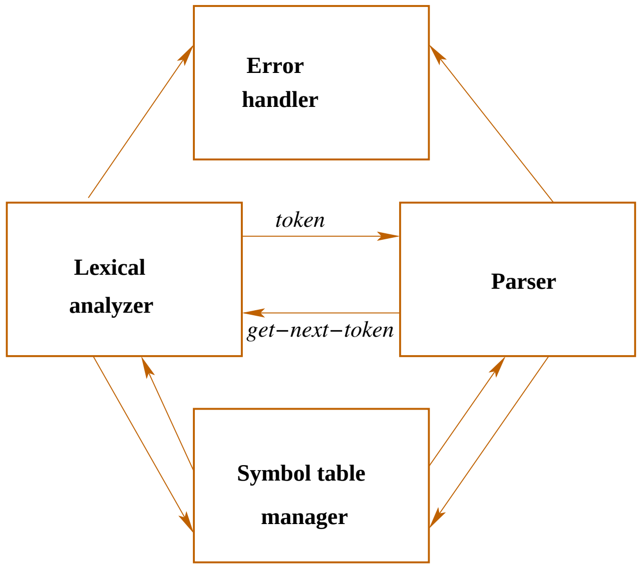


Figure 19: Interaction of Lexical analyzer with parser.

```
<CS>\n          {comments++ ;}

%%

main()    {
               yylex();
               printf("code: %d, comments: %d, white space: %d\n", code, comments, whiteSpace);
              return 0;
}

[moreno@iguanodon lex]$ make C_source_scanner.out
[moreno@iguanodon lex]$ ./bin/C_source_scanner < src/C_source_scanner.l
code: 26, comments: 2, white space: 8
[moreno@iguanodon lex]$
```

2.2 The role of the lexical analyzer in the compiler

Upon receiving a *get-next-tohen* command from the parser, the lexical analyzer reads input characters until it can identify the next token. The lexical analyzer collects also information about tokens into their associated attributes:

- the tokens influence parsing decisions,
- the attributes influence the translation of tokens.

<u>INTERFACING LEX WITH YACC.</u> One of the main uses of LEX is as a companion to the YACC parser-generator.

- YACC expects to call a routine named `yylex()` to find the next input token.
- This routine is supposed to return the type of the next token, as well as putting any associated value in the global variable `yylval`.
- To use LEX with YACC, one specifies the `-d` option to YACC to instruct it to generate the file `y.tab.h` containing definitions of all the tokens appearing in the YACC input. This file is then included in the LEX scanner.

Example 5 *The file* `calculator.y` *below specifies the* <u>GRAMMAR</u> *of the expressions of a desk calculator. This file can be processed by the YACC program.*

```
%{
#include <ctype.h>
#include <stdio.h>

void yyerror(const char *str)
{
  fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
  return 1;
}

main()
{
```

```
  yyparse();
}


%}

%token TOK_NUMBER TOK_PLUS TOK_TIMES TOK_MINUS TOK_DIVIDE TOK_LP TOK_RP

%%
line        : line expr '\n' {printf("%d\n", $2); }
            | /* empty word */
            ;

expr        : expr TOK_PLUS term          {$$ = $1 + $3; }
            | term                  {$$ = $1; }


term        : factor TOK_TIMES term        {$$ = $1 * $3; }
            | factor               {$$ = $1; }


factor      : TOK_LP expr TOK_RP        {$$ = $2; }
            | TOK_NUMBER           {$$ = $1; }
             ;
%%
```

The file `calculator.l` *specifies how the above tokens are detected*

```
%{
#include "y.tab.h"
extern int yylval;
#include <stdlib.h>
%}

PLUS            [\+]
MINUS           [\-]
TIMES           [\*]
DIVIDE          [/]
```

```
DIGIT           [0-9]
NUMBER          [0-9]+
WS              [ \t]*
LP              "("
RP              ")"
RET             [\n]

%%

{WS}            {
                /* eat up white space */
                }
{NUMBER}        {
                yylval = atoi( yytext );
                return TOK_NUMBER;
                }
{PLUS}          {
                return TOK_PLUS;
                }
{TIMES}         {
                return TOK_TIMES;
                }
{MINUS}         {
                return TOK_MINUS;
                }
{DIVIDE}        {
                return TOK_DIVIDE;
                }
{LP}            {
                return TOK_LP;
                }
{RP}            {
                return TOK_RP;
                }
.               {
                }
{RET}           {
                return yytext[0];
```

```
                }
```

The corresponding calculator is built and run as follows:

```
[moreno@iguanodon yacc]$ yacc -d src/calculator.y
yacc: 16 shift/reduce conflicts
[moreno@iguanodon yacc]$ flex src/calculator.l
[moreno@iguanodon yacc]$ gcc lex.yy.c y.tab.c -ll -o calculator
[moreno@iguanodon yacc]$ ./calculator
1 + 3
4
2 * 4
8
(2 + 3) * 6
30
(2 + 3) * (4 + 5)
45
```

<u>About the separation of the lexer from the parser.</u>

- <u>Simplification of design.</u> If the lexical analyzer has removed comments and white space, then the parser's computations will be simpler.
- <u>Efficiency.</u> It is easier to optimize and maintain simple little components than a large sophisticated one.
- <u>Portability.</u> If the lexical analyzer is the only part of the compiler which has to worry about the input file (alphabetic peculiarities) and its support then the compiler portability will be enhanced.

Other possible tasks of the lexical analyzer:

- Correlating error messages from the compiler to the source pogram Indeed, the lexical analyzer may be the only component of the compiler aware of line numbers in the source pogram.
- Macro expansion. Indeed, the tokens associated with a macro can be precomputed.

2.3 The design of an efficient lexical analyzer

When building an automaton from a regular expression r (of length $\mid r \mid$) two options at least are possible.

- Apply Thompson's algorithm to build a NFAIT. Then the number of states is $\mathcal{O}(\mid r \mid)$.
- Apply Thompson's algorithm and the subset algorithm to build a DFA. Then the number of states is $\mathcal{O}(2^{\mid r \mid})$.

Now given a word w of length ℓ deciding whether w is a member of $L(r)$ (the language denoted by r) will cost

- $\mathcal{O}(\ell \mid r \mid)$ with the NFAIT and
- $\mathcal{O}(\ell)$ with the DFA.

We present now a method that minimizes the number of states of a DFA. So this method applies to improve the second option.

2.3.1 Minimal automaton

Equivalence of words modulo a language. Let $L \subseteq \Sigma^*$ be a language over the alphabet Σ. Let $u, v \in \Sigma^*$ be two words (that may belong or not to the language L). We say that the language L separates u et v if

$$\left\{ \begin{array}{l} u \in L \\ v \notin L \end{array} \right. \quad \text{ou} \quad \left\{ \begin{array}{l} u \notin L \\ v \in L \end{array} \right. \tag{14}$$

We say that u et v are equivalent modulo L and we write $u \equiv v \mod L$ if for every $w \in \Sigma^*$ the language L does **not separate** the words $u\,w$ and $v\,w$. In other words we have

$$u \equiv v \mod L \quad \Longleftrightarrow \quad (\forall w \in \Sigma^*)\ (u\,w \in L \quad \Longleftrightarrow \quad v\,w \in L) \tag{15}$$

The relation $u \equiv v \mod L$ is an equivalence relation (since it is reflexive, symmetric and transitive) and thus defines equivalence classes called the *equivalence classes* (or *residue classes*) *associated with* L.

Equivalence of words modulo an automaton. Let $\mathcal{A} = (\Sigma, S, i, F, \delta)$ be a DFA. Let $u, v \in \Sigma^*$ be two words We say that u et v are equivalent modulo $\mathcal{A}$ and we write $u \equiv v \mod \delta$ if after reading u the DFA is in the same

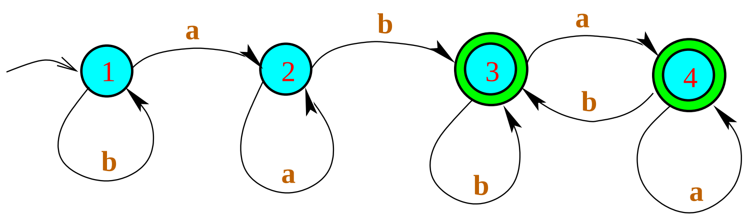


Figure 20: A DFA whith three classes of equivalence.

state than after reading v. Let us denote by $\delta(i,\,u)$ and $\delta(i,v)$ the state of the automaton $\mathcal{A}$ after reading u and v respectively. Then we have

$$u \equiv v \mod \delta \qquad \Longleftrightarrow \qquad \delta(i,u) = \delta(i,v). \tag{16}$$

Again the relation $u \equiv v \mod \delta$ is an equivalence relation and thus defines residue classes associated with the automaton $\mathcal{A}$.

Example 6 *The DFA over $\Sigma = \{a,b\}$ on Figure 20 has three residue classes:*

- *that of the words with no a,*
- *that of the words with no b after an a,*
- *that of the words with a factor ab.*

Proposition 8 *Let $L \subseteq \Sigma^*$ be a language over Σ recognized by the DFA $\mathcal{A} = (\Sigma, S, i, F, \delta)$. For every words u and v we have*

$$u \equiv v \mod \delta \implies u \equiv v \mod L \tag{17}$$

Remark 4 Proposition 8 expresses the fact that every residue class modulo the automaton $\mathcal{A}$ is entirely contained in a residue class modulo its associated language L. Therefore the number of classes modulo L is upper bounded by the number of classes modulo $\mathcal{A}$ and thus by the number of states of $\mathcal{A}$. It follows from the following theorem that having a finite number of residue classes is a necessary and sufficient condition for a language to be recognized by FA.

Theorem 4 *Let $L \subseteq \Sigma^*$ be a language over the alphabet Σ. Let n be a positive integer. If there exist exactly n residue classes modulo L then there exists a DFA with n states that recognizes L.*

MINIMAL AUTOMATON. Let $L \subseteq \Sigma^*$ be a language over the alphabet Σ

- recognized by FA and
- with n residue classes.

It follows from Proposition 8 and Theorem 4 that

- every DFA recognizing L has at least n states,
- there exists a DFA with n states recognizing L.

Therefore we can call *minimal automaton* every DFA with n states and recognizing L.

Remark 5 The number of residue classes of a language is rarely known (until one gets a minimal DFA accepting this language). So one needs a way to characterize *minimal automata*.

Then one needs an algorithm to compute from a given DFA (minimal or not) recognizing L a minimal automaton recognizing L too. Such an algorithm will group states that do *essentially the same job*. This leads to the following notions.

EQUIVALENCE OF STATES IN A DFA. Let $\mathcal{A} = (\Sigma, S, i, F, \delta)$ be a DFA. We say that two states $p, q \in S$ are *equivalent* and we write $p \equiv q$ if for every word $w \in \Sigma^*$ we have

$$(\delta(p, w) \in F) \iff (\delta(q, w) \in F). \tag{18}$$

Let k be a non-negative integer. We say that two states $p, q \in S$ are *equivalent at order k* and we write $p \equiv_k q$ if for every word $w \in \Sigma^*$ with length $\mid w \mid$

$$(\mid w \mid \leq k) \quad \Longrightarrow \quad ((\delta(p, w) \in F) \quad \iff \quad (\delta(q, w) \in F)) \tag{19}$$

CHARACTERIZATION OF A MINIMAL AUTOMATON. Among other properties, Theorem 5 states that for a minimal automaton, each residue class of states contains only one state.

Theorem 5 *Let $L \subseteq \Sigma^*$ be a language recognized by FA and let n be its number of residue classes. Let $\mathcal{A} = (\Sigma, S, i, F, \delta)$ be a minimal DFA recognizing L. Then we have the following properties*

- *for every words* $u, v \in \Sigma^*$ *we have* $u \equiv v \mod \delta \iff u \equiv v \mod L$.

- *for every* $q \in F$ *there exists a word* $u \in \Sigma^*$ *such that* $\delta(i, u) = q$.

- *for every states* $p, q \in S$ *we have* $p \equiv q \iff p = q$.

CONSTRUCTION OF THE EQUIVALENCE CLASSES FOR STATES. Proposition 9 suggests an algorithm for constructing the equivalence classes of states.

Proposition 9 *Let* $\mathcal{A} = (\Sigma, S, i, F, \delta)$ *be a DFA and* k *be a non-negative integer. If the relations* $\equiv_k$ *and* $\equiv_{k+1}$ *have the same residue classes then* $\equiv_k$ *and* $\equiv$ *have the same residue classes too.*

The method is follows.

- Compute $\equiv_0$ which consists of two classes:
  - the set of the final states,
  - the set of the non final states.

- Compute $\equiv_1$ and set k to 0

- Whhile $\equiv_k$ and $\equiv_{k+1}$ are different $\equiv_{k+2}$ and set k to $k+1$.

In practice one construct a table that gives the transition from each state after reading any word of length 0, any word of length 1, any word of length 2, etc... In these tables we underline final states in order to detect classes more easily. Quite often computations can be simpler. For instance, in Example 7 computations can be stopped after computing the third column. Explain why!

THE CONSTRUCTION OF A MINIMAL AUTOMATON $\overline{\mathcal{A}} = (\Sigma, S', i', F', \delta')$ from a given DFA $\mathcal{A} = (\Sigma, S, i, F, \delta)$ can be down as follows (where the class of a state $s \in S$ is denoted by $\overline{s}$).

- Compute the equivalence classes of states. From Theorem 5 they provide S', also denoted by $\overline{S}$.

- Let $s, s' \in S$ such that $\overline{s} = \overline{s'}$. Let $x \in \Sigma$. We have $\overline{\delta(s, x)} = \overline{\delta(s', x)}$. Hence we can define $\delta'(\overline{s}, x)$ as

$$\delta'(\overline{s}, x) = \overline{\delta(s', x)} \tag{20}$$

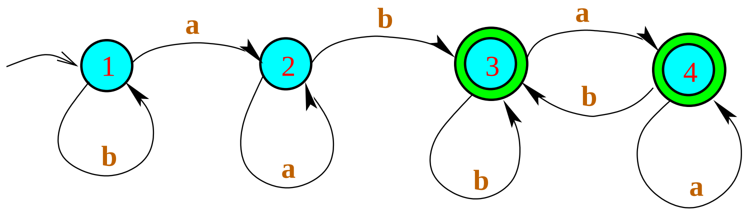


Figure 21: A DFA whith three classes of equivalence.

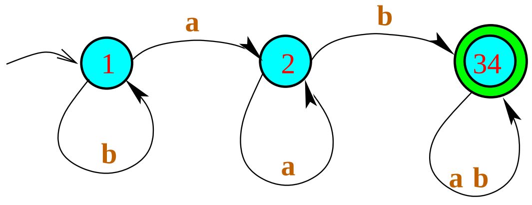


Figure 22: Minized automaton.

- We choose $i' = \overline{i}$.
- We define $F' = \{\overline{s} \mid s \in F\}$.

Example 7

ε	a	b	aa	ab	ba	bb
1	2	1	2	1	2	1
2	2	$\underline{3}$	2	$\underline{3}$	$\underline{4}$	$\underline{3}$
$\underline{3}$	$\underline{4}$	$\underline{3}$	$\underline{4}$	$\underline{4}$	$\underline{4}$	$\underline{3}$
$\underline{4}$	$\underline{4}$	$\underline{3}$	$\underline{4}$	$\underline{4}$	$\underline{4}$	$\underline{3}$

Example 8

ε	a	b	aa	ab	ba	bb
1	2	6	7	$\underline{3}$	$\underline{3}$	7
2	7	$\underline{3}$				
$\underline{3}$	1	$\underline{3}$	2	6	1	$\underline{3}$
5	8	6	7	$\underline{3}$	$\underline{3}$	7
6	$\underline{3}$	7	1	$\underline{3}$	7	$\underline{3}$
7	7	5	7	5	8	6
8	7	$\underline{3}$				

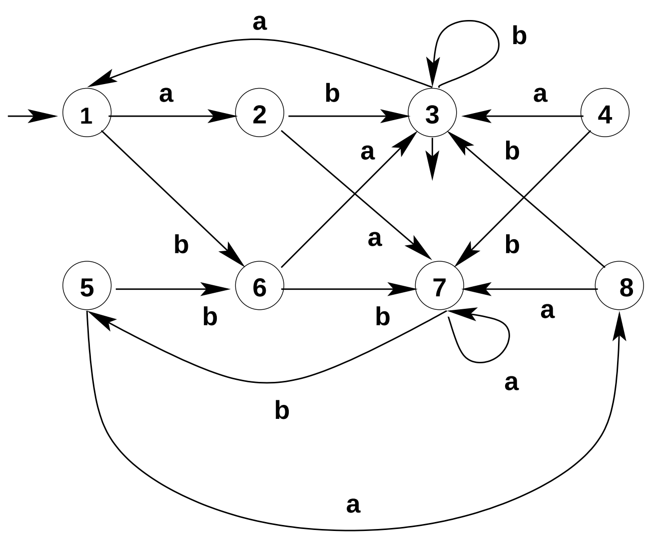


Figure 23: Another DFA to minize.

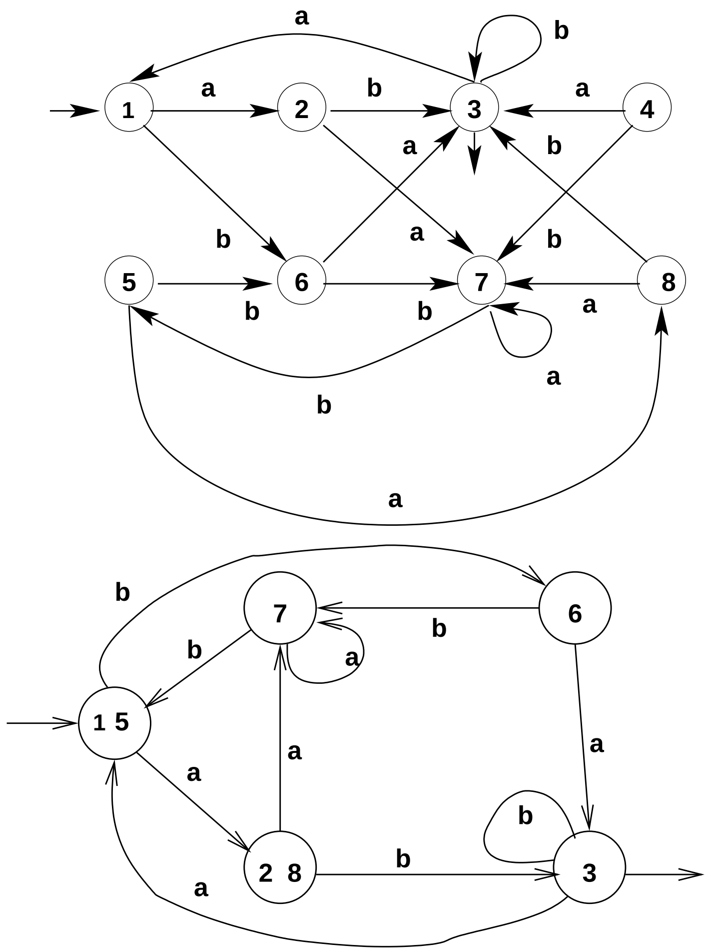


Figure 24: Minized Automaton.